Generation of Efficient High-Level Hardware Code from Dataflow Programs

Nicolas Siret, Matthieu Wipliez, Jean François Nezan, Francesca Palumbo

Generation of Efficient High-Level Hardware Code from Dataflow Programs

Nicolas Siret and Matthieu Wipliez and Jean-François Nezan
European university of Brittany, France
INSA, IETR, UMR 6164, F-35708 RENNES
Email: name.surname@insa-rennes.fr

Francesca Palumbo
University of Cagliari, Italie
Email: francesca.palumbo@diee.unica.it

***Abstract*—High-level synthesis (HLS) aims at reducing the time-to-market by providing an automated design process that interprets and compiles high-level abstraction programs into hardware. However, HLS tools still face limitations regarding the performance of the generated code, due to the difficulties of compiling input imperative languages into efficient hardware code. Moreover the hardware code generated by the HLS tools is usually target-dependant and at a low level of abstraction (i.e. gate-level). A generated code at a high-level of abstraction (i.e. chip-level) is better suited to the needs of systems' architects because they can understand and control all of the design processes. We propose in this paper a new approach to HLS to generate efficient, high-level hardware code from Dataflow Programs. Implementation results (from two dynamic dataflow programs) on Xilinx, Altera and Latice FPGAs and on ASIC targeting 90nm CMOS technology are also presented.**

I. Introduction

High-Level Synthesis (HLS) aims at reducing both the complexity and the time-to-market of new applications on hardware architectures. HLS is currently based on System- or Model-Level (e.g. C, SystemC.) programming, and target-specific (e.g. VHDL, System Verilog) code generation. Programming at a higher level of abstraction allows designers to abstract usual low-level technicalities associated with hardware description languages. The usual low-level technicalities are managed by the HLS tools which analyse the input designs and insert the logic required to ensure a correct behaviour. Not only does it result in higher productivity, but it also increase performance by speeding up the design flow and thus providing more opportunities for debugging and performance tuning.

There has been a large body of research and development on HLS [3], [4], leading to the emergence of third-generation HLS tools either sold by major companies (e.g. Synopsis, Mentor Graphics), or provided "as is" (e.g. OpenCores). The suppliers of HLS compilers [4] and independent benchmarkers [5] compared the HLS to the usual hand-coded development stages (i.e. coding, debugging, implementing, validating) for both hardware and software programming. According to their results, using HLS reduces the time-to-market while keeping or improving the RTL quality and the final performance of a design.

However HLS tools still face limitations, especially to extract a flexible and efficient hardware code from sequential algorithms. This paper presents the usual limitation of HLS tools and proposes a new two-step approach for implementing HLS in a compilation infrastructure. The first step consists in compiling the applications into an efficient, high-level and portable hardware code; the second step in generating the RTL code and the bitstream using the usual hardware synthesizers. The hardware code is generated according to a good coding style which allows synthesizers to perform optimizations and which eases the code refinement. Applications are described using a dataflow programming language based on the DataFlow Process Network Model of Computation (MoC) [1].

This paper makes the following contributions with respect to the research presented in [2] on hardware code generation:

- we present the transformations and modifications performed to implement the two-step HLS in the Open RVC-CAL Compilation infrastructure,
- we show in section III how to generate a hardware code which, (1) can be synthesized on various FPGAs and ASIC technologies with good performance, and (2) is easily understandable and maintainable,
- we detail the optimizations made to reduce the required area, especially on the inter-entities communication protocol (section IV).

Performance assessment of the presented two-step HLS is going to be performed on a wide set of targets: a Xilinx Spartan3 FPGA, an Altera Stratix-V FPGA, a Latice ECP2 FPGA and also an ASIC using a 90nm CMOS technology. All those targets run two different designs extracted from two different dynamic dataflow programs: a 2D Inverse Discrete Cosine Transform and an AC/DC prediction (section V).

II. Background

This section presents related work on High-Level Synthesis and our approach compared to others.

A. Related Work

As presented in the fig.1, HLS tools allow designers to program at a higher level of abstraction to avoid complex low-level debugging using cycle accurate simulator (e.g. Modelsim). In many cases the source programs are described using System- or Model-level languages, like SystemC [6] because it mimics some aspects of hardware-oriented language while

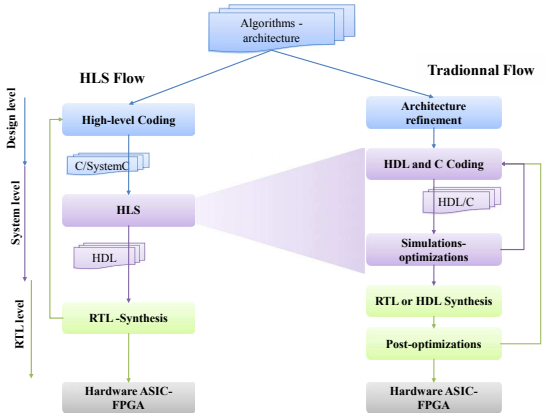

Fig. 1: *High Level Synthesis* Design flow.

having the same expressiveness as a software language, or C because it is one of the most used in the industry. However limitations still restrict the compilation of sequential source programs:

- the compilation is often not fully automated[1] [7], [8], because of the considerable number of possibilities offered by the hardware code compilation. Designers have thus to make multiple choices on specific elements (e.g. mapping, technological target).
- the imperative nature of the input languages [4], [8], restricts the performance of the generated code with respect to the slices used and to the throughput. Indeed, the lack of inherent parallelism in the source language means that the generated design will execute in more cycles than necessary.
- byte-based types used in the majority of software languages make it difficult for HLS compilers to minimize resources.

In addition, the code generated from sequential sources programs have their own limitations:

- the generated code is mostly at a low-level of abstraction (i.e. gate-level) which does not meet the need of designers used to code at a higher level of abstraction (i.e. chip-level). As a result, designers have difficulty analysing this generated code and thus optimizing their input designs to increase the performance.
- generating a design at a low-level of abstraction to ensure the same performance whatever the design, actually restricts the optimizations made by all recent hardware synthesizers because they expect synthesizing hand-coded hardware code at a medium- or high-level of abstraction.

B. Our approach to HLS

As presented in the introduction, we enhanced the work presented in [2] and developed a hardware compilation infrastructure which implements a two-step HLS that fill the gap between HLS and usual hand-coded methodologies. In concrete terms, the two-step HLS we propose consists in (*first step*) compiling a dataflow programs into hardware code while keeping as many similarities as possible from the source; and (*second step*) letting the synthesizers make the optimizations usually made by HLS tools (e.g. removing unused gates, optimizing the critical path).

The hardware compilation infrastructure is implemented into the Open RVC-CAL Compiler [1] (Orcc). Orcc is a multi-targets compilation infrastructure written in Java[2] which allows compiling dataflow programs into various target languages such as C [10], LLVM [11], etc. The code compilation is processed in two steps: (1) the front-end compiles dataflow programs into an Intermediate Representation (IR) and (2) the IR is transformed into a target language. Mono-core, multi-cores and mixed hardware/software architectures can be targeted using Orcc.

The dataflow programs we consider are provided in the Reconfigurable Video Coding (RVC) framework [9]. The MOC define RVC-CAL programs as hierarchical block diagrams called *networks*, where blocks can be either *networks* or *actors*, and communicate with each other through unidirectional FIFO channels. An RVC-CAL actor may have input and output ports, parameters, variables, functions and procedures, *actions* that may be identified by a tag, a Finite State Machine (FSM), and a set of priorities that establish a partial order between action tags. The behaviour of an actor is defined within its actions which can consume or produce tokens, and process algorithms. The time spent to learn a new language is balanced by the time saved on the development stages, considering this learning is required just once.

The novel HLS approach presented in this paper and depicted in fig.2 provides an answer to the usual limitations of HLS tools because:

- Using the RVC-CAL dataflow language solves the problem of efficiently extracting parallelism from imperative language since RVC-CAL naturally highlights the parallelism of applications and empowers designers with the ability to describe inherently parallel applications.
- The restrictions previously introduced in terms of code portability, reuse and refinement are overcomed by the two-step HLS, closer to the traditional hardware development flow. Moreover, this code is efficiently (in terms of performance, area, etc.) and equally well synthesized on the various hardware synthesizers.

III. Implementation of the two-step HLS

We present in this section, the requirements needed to implement the two-step HLS in the Orcc compilation infrastructure.

A. Basic Rules and Motivations

In our two-step HLS, three rules must be validated: (1) novices and senior designers must be able to easily use the

[1]Some of the latest commercial tools, partially overcomes this limitation [4] using complex algorithms that can perform several tests to find the best choice in terms of performance, area or a balance between the two.

[2]Orcc is available as a feature for the Eclipse environment, see orcc.sf.net

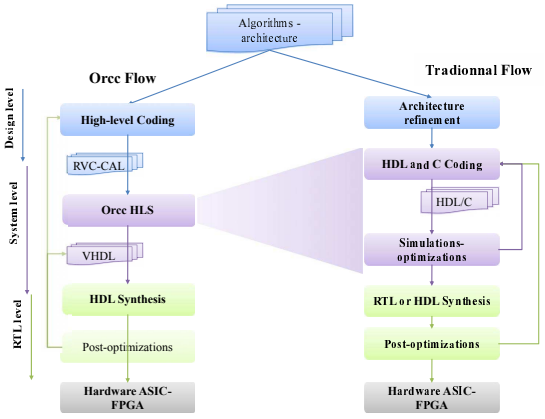

Fig. 2: Open RVC-CAL Compiler design flow.

compiler, (2) the generated code must be flexible and maintenable, (3) the generated code must provide good performance no matter what technology is targeted (i.e. FPGAs, ASICs). Rule (1) is validated by the compilation infrastructure (Orcc) which has been provided as an Eclipse plug-in to guarantee usability, and by the RVC-CAL language which is halfway between hardware and software languages.

The only way to validate rules (2) and (3) is to follow a set of rules, also known as good synthesis *coding style*. FPGAs, ASICs and IP providers supply their own coding style [12], [13] whose essence is to ensure higher performance by avoiding writing code that creates useless complexity and gates amount. The portability between FPGA families is obtained by a mix of these coding styles in addition with others specific coding rules defined in particular by IP providers [14]. Following a good synthesis coding style is crucial using high-level VHDL code and it is clearly declared by FPGAs providers like Xilinx: *"certain seemingly minor decisions made while crafting an RTL-level design can mean the difference between a design operating at less than 100 MHz and one operating at more than 400 MHz"*.

B. Actors' body coding style

The fig. 3a presents one of the action (i.e. *limit*) of the"Clip" actor which performs a clipping operation: if the input token has a value greater than 255 or less than 0, the token is clipped to 255 or min respectively. The value of min is determined in another action. A part of the VHDL code compiled from this action is presented in the fig. 3b, the generated code is naturally understandable and can easily be compared to the RVC-CAL one.

An FPGA is composed of logic elements whose interconnection are programmable so as to carry out different functions as required by the design. These logic elements can be binary operators (e.g. *and*, *or*), arithmetic operators (*add*, *sub*), memories (e.g. flip-flops, latches), multiplexers, etc. Latches must be avoided because they cause instability and lengthen the critical path. In this way, we removed al the latches by a proper use of the VHDL variables and signals:

```
actor Clip ()
  int(size=10) I, bool SIGNED ==> int(size=9) O :

  int(size=7) count := -1;
  bool sflag;

  limit: action I:[i] ==> O:[ if i > 255 then 255
    else if i < min then min else i end end ]
  var
    int min = if sflag then -255 else 0 end
  do
    count := count - 1;
  end

  //..
end
```

(a) The `limit` action of the `Clip` actor in RVC-CAL.

```
Xilinx_clip_execute : process (reset_n, clock) is
  -- variable declaration (...)
begin
  if reset_n = '0' then
    -- (...)
  elsif rising_edge(clock) then
    -- (...)
    elsif (limit_go = '1') then
      -- body of "limit" action
      limit_local_sflag := sflag;
      limit_local_count := count;
      limit_I_i         := to_integer(signed(I_data));
      if (limit_local_sflag = '1') then
        limit_tmp_if := -255;
      else
        limit_tmp_if := 0;
      end if;
      limit_min_1       := limit_tmp_if;
      limit_local_count := limit_local_count - 1;
      -- (...)
    end if;
  end if;
end process Xilinx_clip_execute;
```

(b) Part of the `limit` action of the `Clip` actor in VHDL.

Fig. 3: Actor code generation: core of the actor.

all the tokens to be memorized are stored in signals and the computations are performed on variables (which temporary contain the token) before returning the results. The name and the type of the VHDL variables and signals are extracted from the RVC-CAL programs.

The MOC allows an actor to contain several actions however it also defines that only one action can be executed per cycle in an actor. As a consequence, all actions are combined in a single sequential process which make the behaviour of the code predictable (the designer knows exactly which action is executed in the current cycle) without impacting either the number of slices required or the critical path.

C. Actors' scheduler coding style

An RVC-CAL actor has *actions* which define its behaviour, these actions may be identified by tags which can be ordered using a Finite State Machine (FSM), and a set of priorities. This ordering defines the scheduling of the actions within an actor. In our approach, the scheduler manages the actions scheduling and the transmission of the token reception acknowledgement between actors using an {if(), elsif(), end if} structure. The fig. 4 shows a part of the VHDL "Clip" actor scheduler. The generated scheduler:

```
clip_scheduler : process(I_send, SIGNED_send,
                         O_rdy, count)
  -- variable declaration (...)
begin
  -- (...)
  -- test if "limit" action is schedulable
  isSchedulable_limit_local_count_1 := count;
  if (isSchedulable_limit_local_count_1 >= 0) then
    isSchedulable_limit_result_1 := '1';
  else
    isSchedulable_limit_result_1 := '0';
  end if;
  isSchedulable_limit_go := isSchedulable_limit_result_1;
  -- (...)
  limit_go <= '0';
  I_ack <= '0';
  -- (...)
  elsif((isSchedulable_limit_go and I_send ) = '1') then
    limit_go <= isSchedulable_limit_go and O_rdy;
    I_ack <= isSchedulable_limit_go and O_rdy;
  end if;
end process;
```

Fig. 4: Parts of the Schedule of the `Clip` actor in VHDL.

- may check if tokens are available in the port of an actor (i.e. *I_send* in the instance), since actions can only be fired if one or more tokens are available on its input ports,
- may control if a token can be sent to the target actor (i.e. *O_rdy* in the instance),
- may send an acknowledgement (i.e. *I_ack* in the instance) to a source actor when a token is consumed.

The scheduling of an action is usually made on the main process which is clocked. However, in a code generated from dynamic dataflow designs (in which sending an acknowledgement when a token is consumed is compulsory), it makes the throughput decrease from an action executed per cycle to an action every two. In our approach shown in the fig.5, the scheduler is executed in a combinatorial process activated by the transmission signals to ensure the processing of one action per cycle Using a combinatorial process split from the

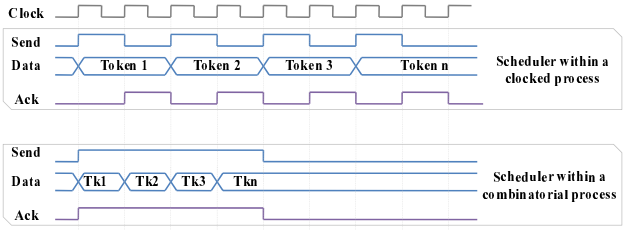

Fig. 5: Throughput depending on the location of the scheduler.

main sequential process may slightly increase the critical but ensure a throughput of one token per cycle.

D. Inferring dual-ports RAM

Portability is strategic for hardware IP vendors since their customers may use different FPGA and ASIC families. Besides, generating and maintaining a code non-synthesizable on, at least, Xilinx and Altera FPGAs is clearly unproductive. In usual code generators, designers must provide a template of the internal FPGA RAM to allow the HLS which makes the code platform dependant. We dealt with this problem and decided to adopt vendor-neutral RAM entities, forcing synthesizers to infer them on the specific RAM components of the target device. Inferring a RAM no matter the FPGA and technologies also required to respect a good synthesis coding style. Thereby, in the two-step HLS the lists (arrays) are transformed either into register (small lists) or internal FPGA RAM (medium and large lists).

IV. Inter-actors communication

The communication protocol between actors is also important in our two-step HLS. Indeed, hardware code generated from a dataflow programs necessarily suffer a inter-actors communication overhead with respect to hand-written codes. This overhead must naturally be as low as possible. At the same time, the protocol must be printable and close to manual coding (first step of our approach) and it must allow synthesizers to perform optimizations (second step).

A. Source of Communication Overhead

Hardware designs, generated from dynamic dataflow programs, are affected by an inter-entity communication overhead to behave as hand-written designs. The reason is the model behind dynamic dataflow programs states that both actors production and consumption are not known a priori, i.e. actors can receive and send data at any rate. Whereas, dealing with manual hardware implementations, entities produce and consume data at a fixed rate. This is not the case with manual hardware implementations, in which entities produce and consume data at a fixed rate. In our case, a generated entity may need to wait for predecessors (resp. successors) before being able to read (resp. write) data on its input (resp. output) ports. This means that the generated code for an entity must include code to control how data is transmitted to other entities.

To better understand where the communication overhead comes from, we consider the model of the Inverse Discrete Cosine Transform (IDCT) shown in the fig. 6. This IDCT model is provided along with all the ratios of tokens production and consumption on the communication buses. In this

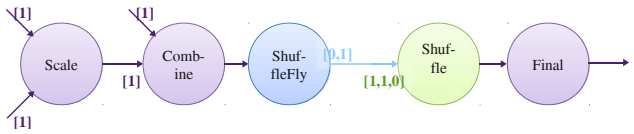

Fig. 6: A model of *idct* with the ratios of tokens productions and consumption.

model of *idct*, the ShuffleFly actor produces one token every two cycles [0, 1] while the Shuffle consumes a token every two three-cycle [1, 1, 0]; the other actors produce and consume a token every cycle [1]. The control of transmission between the ShufflyFly and the Shuffle actors is thus compulsory to avoid errors of transmission [$0 \rightarrow 1$ (error), $1 \rightarrow 1$, $0 \rightarrow 0$, ...].

B. Minimization of the Communication Overhead

The overhead reduction problem is not trivial. It is necessary to conceive and implement a clever control for data transmission to minimally affect the area and the throughput of the automatically generated hardware design. On most of the HLS tools, the data transmission managed leveraging on FIFOs, or memories, instantiated between any two instances of a generated network. However, experience shows this is not a satisfactory solution because:

- it needlessly complicates the code of the hardware networks,
- it can increase the number of logical slices used,
- it can increase the power consumption of the design.

To overcome the above mentioned issues, we develop two optimized hand coded IPs, shown in fig. 7, which handle the communication protocol: (1) a communication arbiter (*comArbiter*) and (2) a broadcast manager (*broadcast*). These two IPs always know if tokens are available or if the bus is free between two actors and inform the actor scheduler (*O_rdy* in the instance presented, section III-C).

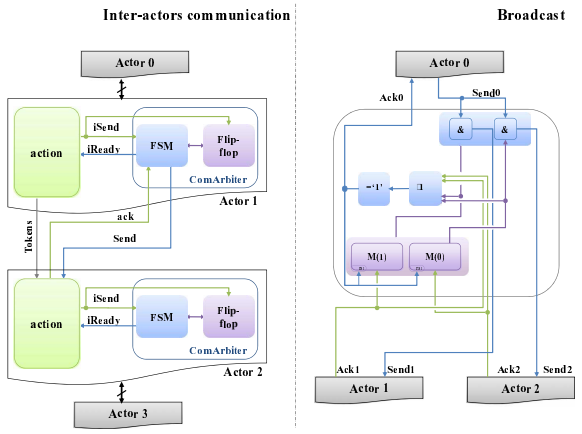

Fig. 7: the communication arbiter and the broadcast manager.

The communication arbiter is composed of an FSM and a single bit memory block (usually a Flip-Flop). The Flip-Flop stores the information of tokens production from the actor. The FSM always checks the state of the network (if a token is passed on) and the actor (if a token is available) and informs the actor scheduler. When the network is free (a token can be sent to the next actor), it unlocks the actor scheduler, otherwise it locks the actor scheduler to prevent the token from being lost. Note that the data is never stored in a the *comArbiter* which reduces the required implementation area to nearly zero without influence on the critical path.

Similarly, the broadcast managing all the communications one to many, is made up of logic cells and single bit memory blocks (usually Flip-Flops). The Flip-Flops store the information of data consumption, and the logical cells manage the control of the transmissions within the broadcast. As long as a token is not consumed by one of the target actors, the *Input send* information is passed (logical `and` between the token consumption acknowledgement and the send). The token consumption acknowledgement *Input ack* is sent to the source actor when all the targeted actors have consumed the token (logical `equal` between all the token consumption acknowledgement and "11...1"). The fig. 7 shown an example of a broadcast of size two but the IP is coded using generic parameters so as to be compliant with all the sizes of broadcast.

V. Results

The performance of the presented two-step HLS has been assessed on typical components of video decoders: the 2D Inverse Discrete Cosine Transform (IDCT) and the AC/DC prediction. In order to show the portability of the generated hardware codes we are going to discuss the achieved results both on different FPGAs vendors' platforms, namely an Altera StatrixIII, a Xilinx XC3S4000 and a Diamond LFE2 FPGAs, and also targeting an ASIC design flow adopting a 90nm CMOS technology. The Xilinx FPGA, Lattice FPGA and Altera FPGA are respectively a low-, medium-, and high-cost/performance FPGAs. The behaviour of the IDCT and the AC/DC generated codes have been validated using Modelsim 6.6 and the synthesis have been performed using respectively Altera QuartusII, Xilinx ISE, Lattice Diamond and the Design Compiler tool of Synopsis.

For the sake of comparison, we have looked for proprietary IPs and/or open source codes; unfortunately none of them is free of charge, even for research purposes. In the IDCT case, we succeeded in finding in literature an implementation on the Xilinx XC3S4000 [15], from now on referred as (@IP). Finally both the use cases have been also synthesized in hardware, from their dynamic dataflow models, using OpenForge [16]. All the synthesis results are summarized in the table proposed in the fig.8.

A. Performance on the 2D Inverse Discrete Cosine Transform

In the FPGA case, the highest frequency is obtained using the @IP. Nevertheless the approach presented in this paper, at the price of having an operating frequency, 8% lower compared to this fully optimized IP ("low Power, high Speed DCT/IDC"), is able to benefit from 60% less Slices, and 23% less LUTs. Unfortunately, the work in [15] does not present other reference platforms besides the Xilinx XC3S4000 one and OpenForge has been developed for Xilinx as well; therefore the performance on Altera and Lattice do not have any direct comparison. With respect to OpenForge the presented approach provides better results for all the considered indexes. In the ASIC case, the Synopsys Design Compiler allowed us to carry out two types of synthesis: (1) without stringent timing constraints (area, timing and power have been treated in the same way) or (2) privileging timing with respect to area and power. In the former it was possible to guarantee an operating frequency of 714MHz with an area occupation of 0.21mm2. In the latter case instead it was possible to reach 1GHz but an area occupation of 0.22 mm2.

	XILINX (XC3S4000)				ALTERA (EPS3SL50)				LATICE (LFE2-50E)				ASIC	
	Fmax (MHz)	Slices	LUT	Power (mW)	Fmax (MHz)	Logic	ALU	Power (mW)	Fmax (MHz)	Slices	LUT	Power (mW)	Fmax (MHz)	Area (mm2)
2D Inverse Discrete Cosine Transform														
Orcc-HDL	52	1154	1794	17, 84	160	1079	1418	42, 69	70	1011	828	13 ,3	1000	0, 22
@ IP	55	3571	4640											
OpenForge	41	1413	3584	23, 21	X	X	X	X	X	X	X	X	X	X
AC/DC Prediction														
Orcc- HDL	63	926	2119	9, 94	124	804	1355	75, 75	51	984	1244	7, 4	625	0, 20
OpenForge	92	957	2118	15,6	X	X	X	X	X	X	X	X	X	X

Fig. 8: Performance of the generated code using HLS in Orcc.

B. Performance on the AC/DC prediction

The fig.8 also shows the implementation results on an AC/DC prediction network composed of 7 actors. Our approach provides attractive results, no matter the FPGA or ASIC technology.

VI. Conclusion

This paper proposes a new approach to High-Level Synthesis to generate efficient, high-level, portable hardware code starting from dynamic dataflow programs in two steps. The compilation and implementation results show that:

- the code is generated once and is portable on FPGAs and ASICs, from the low-cost ones, to the high-cost and efficient ones.
- the refinement, understandability and reuse are facilitated thanks to a code generated at high-level (i.e. chip-level) of abstraction rather than low-level (i.e. gate-level) abstraction.
- Hardware synthesizers can perform optimizations (loop managing, optimization of critical path) that can not be achieved with low-level abstraction code,
- performance of the generated code matches that of the hand-coded hardware in terms of frequency, power and area.

Thereby, this approach supply a solution to overcome the usual restrictions of HLS tools, in terms of portability, performance or reusability.

Many interesting areas for future research involve generating low-power or low-area design, and translating higher-level RVC-CAL constructs such as for-loops and multi-token reads and writes into optimized high-level VHDL code. Another interesting area of research involves generating both hardware and software code to target heterogeneous platforms such as Armadeus Systems' APF-51. Based on our previous experience [17] we also foresee the possibility of automating hardware/software co-design for this kind of platform.

References

[1] Wipliez, M, "Compilation Infrastructure for Dataflow Programs," Ph.D. dissertation, National Institute of Applied Sciences (INSA) - Rennes, 2010.

[2] N. Siret, M. Wipliez, J. Nezan, and A. Rhatay, "Hardware code generation from dataflow programs," in *IEEE International conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2010, pp. 113 –120.

[3] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, pp. 18–25, 2009.

[4] P. Coussy and A. Morawiec, *High-level synthesis: from algorithm to digital circuit*. Springer Verlag, 2008.

[5] I. Berkeley Design Technology, "An independent evaluation of: High-level synthesis tools for xilinx fpgas," www.BDTI.com, Tech. Rep., 2010.

[6] *IEEE Std 1666 - IEEE Standard SystemC Language Reference Manual*, IEEE Std 1666-2005, 2005.

[7] J. Castillo, P. Huerta, and J. Martínez, "An Open-Source Tool for SystemC to Verilog Automatic Translation," in *Latin American Applied Research*, 2007.

[8] P. Coussy, D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," *Design Test of Computers, IEEE*, vol. 26, no. 4, pp. 8 –17, 2009.

[9] M. Mattavelli, I. Amer, and M. Raulet, "The Reconfigurable Video Coding Standard [Standards in a Nutshell]," *IEEE Signal Processing Magazine*, vol. 27, no. 3, pp. 159–167, may 2010.

[10] M. Wipliez, G. Roquier, and J.-F. Nezan, "Software Code Generation for the RVC-CAL Language," *Springer journal of Signal Processing Systems*, 2009.

[11] J. Gorin, M. Wipliez, J. Piat, F. Preteux, and M. Raulet, "An llvm-based decoder for mpeg reconfigurable video coding," in *IEEE Workshop on Signal Processing Systems (SIPS)*, 2010.

[12] Altera, *Recommended HDL Coding Styles*. Altera, 2010.

[13] Xilinx, *Coding Style Guidelines*. Xilinx, 2010.

[14] J. ASHENDEN, Peter, *The Designer's Guide to VHDL - third edition*. Morgan Kaufmann Publishers, 2008.

[15] R. Megalingam, K. Venkat, S. Vineeth, M. Mithun, and R. Srikumar, "Hardware Implementation of Low Power, High Speed DCT/IDCT Based Digital Image Watermarking," in *International Conference on Computer Technology and Development (ICCTD)*, vol. 1, nov. 2009, pp. 535 –539.

[16] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing Hardware from Dataflow Programs," *Journal of Signal Processing Systems*, 07 2009.

[17] N. Siret, I. Sabry, J. Nezan, and M. Raulet, "A codesign synthesis from an MPEG-4 decoder dataflow description," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2010, pp. 1995–1998.